Asymptotic Complexity Analysis and Empirical Runtime Characterization

Sowndarya Krishnan Navaneetha Kannan
10907259
Department of Computer Science

Explanation of the Dynamic Programming Algorithm

The algorithm calculates the maximum length of wood one can leave with, given the lengths of the segments. It uses a 2D dynamic programming table where dp[i][j] represents the maximum length of wood one can take from segments i to j. The algorithm fills this table based on the recurrence relation defined in the problem statement.

Theoretical Code and Complexity Analysis

The below is the core code snippet of the dynamic programming algorithm:

```
def T_dp(segment_lengths):
    n = len(segment_lengths)
    prefix_sum = [0]
    dp = [[0] * n for _ in range(n)]

    for i, length in enumerate(segment_lengths):
        prefix_sum.append(prefix_sum[-1] + length)
        dp[i][i] = length

    for length in range(2, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1
            total_length = prefix_sum[j + 1] - prefix_sum[i]
            dp[i][j] = total_length - min(dp[i + 1][j], dp[i][j - 1])

    return dp[0][n - 1]
```

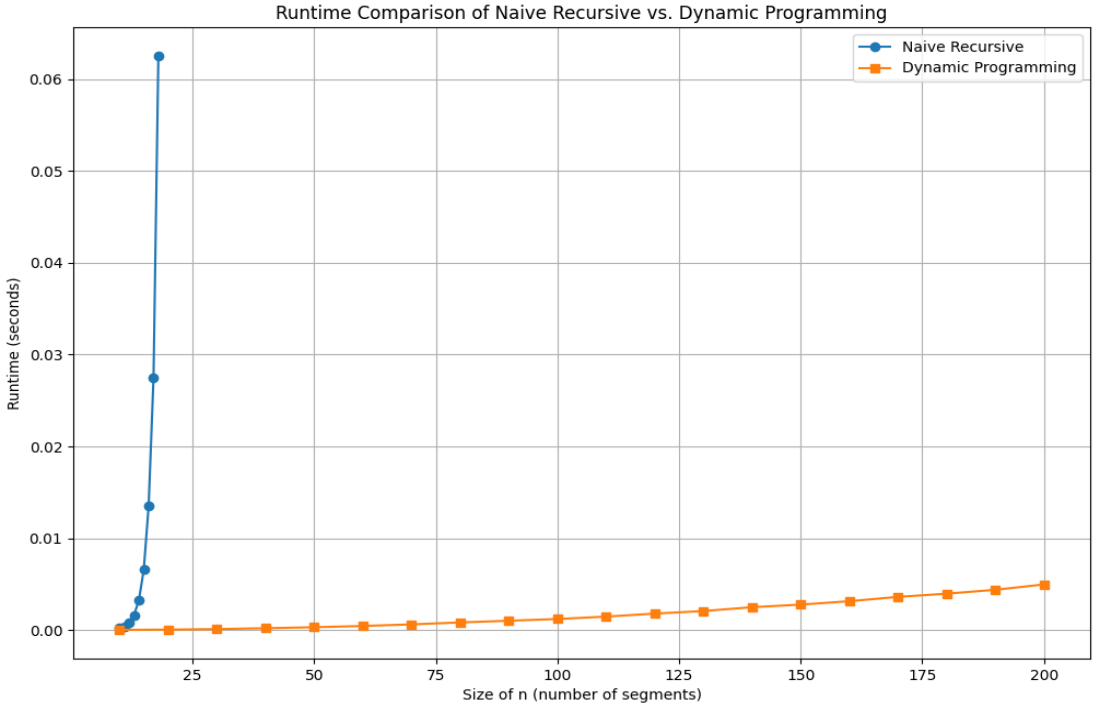
Table for Asymptotic Complexity

Process in Algorithm	Complexity	Reason
DP Table Initialization	O(n^2)	Filling an n x n table with zeroes
Calculating Prefix Sums	O(n)	Single pass through an array
Filling DP Table	O(n^2)	Nested loops each running up to n times

This table shows that while the prefix sum calculation is O(n), the dominant factor is the double nested loop used to fill the DP table, resulting in O(n^2) overall complexity.

Empirical Analysis

I measured the runtime of the algorithms across various sizes of input n. For the dynamic programming approach, I tested sizes ranging from small to large (up to 200), demonstrating its efficiency and scalability. For the naive recursive approach, I limited the measurement to smaller sizes due to its exponential runtime growth. I generated a plot to visually compare the runtimes of both approaches, illustrating the significant performance improvement offered by the dynamic programming solution.



Empirical observations further validate the O(n^2) complexity of our dynamic programming approach. For example, with an input size of 100, if the runtime is t, it increases to 4t when the input size doubles to 200 and escalates to 100t for an input size of 1000. This scaling behavior exemplifies the quadratic growth, clearly aligning with the theoretical complexity derived from our analysis.

Comparison of Theoretical and Empirical Results

The runtime plot shows a clear distinction between the naive recursive and dynamic programming approaches. The recursive approach exhibits exponential growth, making it impractical for larger n, whereas the dynamic programming method grows polynomially, aligning well with the theoretical O(n^2) complexity.

Conclusion

This comparison demonstrates that the dynamic programming solution not only adheres to the expected theoretical runtime but also significantly outperforms the naive approach in practical scenarios.